CSCE230301 - Comp Org.& Assmbly Lang Prog (2020 Summer)

Project 3

Amr Morsy

Mohamed Basuony

Table of Contents

Section 1: Introduction

The goal of this project is to further understand the mechanism behind caches and how they function in the memory hierarchy. In this project, randomly generated numbers were generated and were considered to be addresses. The random addresses were divided to parts that resembled their tags, indices, byte and word offsets. This information was used to determine whether the address should be stored in the cache or it was already there. If the address was already available in the cache, the simulator prints out the word **Hit** next to the address and proceed to the next address. If the address is not stored in the cache, the simulator prints out the word **Miss** and uses the Principle of Locality to fill out the rest of the available places in the cache. The simulator was used to conduct 2 experiments. The first experiment measured how the change in line size affects the cache. The second experiment measured how the change in cache size affected the cache.

Section 2: Design and Implementation

Overview: The program consists of 2 files:

Source.cpp & Direct_Mapping.cpp

In this section, we are going to discuss and explain the design and the implementation of the program by exploring the functions in these 2 files.

Source.cpp: This is the main file. The code in this file is responsible for, primarily, 2 tasks.

Task 1: Generating random addresses. There are 6 functions in the file that produce random addresses with different ranges and values.

Task 2: Passing the random address generated by a specific generator function, as a parameter, to Direct_Mapping.cpp.

In general, Source.cpp generates 1 million random addresses and passes them, as parameters, to Direct_Mapping.cpp.

Direct_Mapping.cpp: This cpp file contains the cache memory, implemented as an array of structures. Each structure corresponds to a cache block. That is, it consist of a valid bit and a tag. These two entires are useful in determining whether the memory access is a hit or a miss, according to the address passed from Source.cpp. There are 2 main functions.

Check_Cache: This function, primarily, does 2 tasks.

Task 1: Derive the index and the tag. This is done by performing specific

Logical shift right to the address.

Task 2: Using the index derived in task 1, it checks the cache array if the valid bit of the this cache block is set or not. If it is set, it checks if there is a matching tag. If that is the case, then the function returns true. Otherwise, it returns false.

Add_To_Cache: This function does only one task; It sets the valid bit and allocates the tag derived in Check_Cache in the cache block. This function is instantiated in case of misses.

Section 3: Experiments

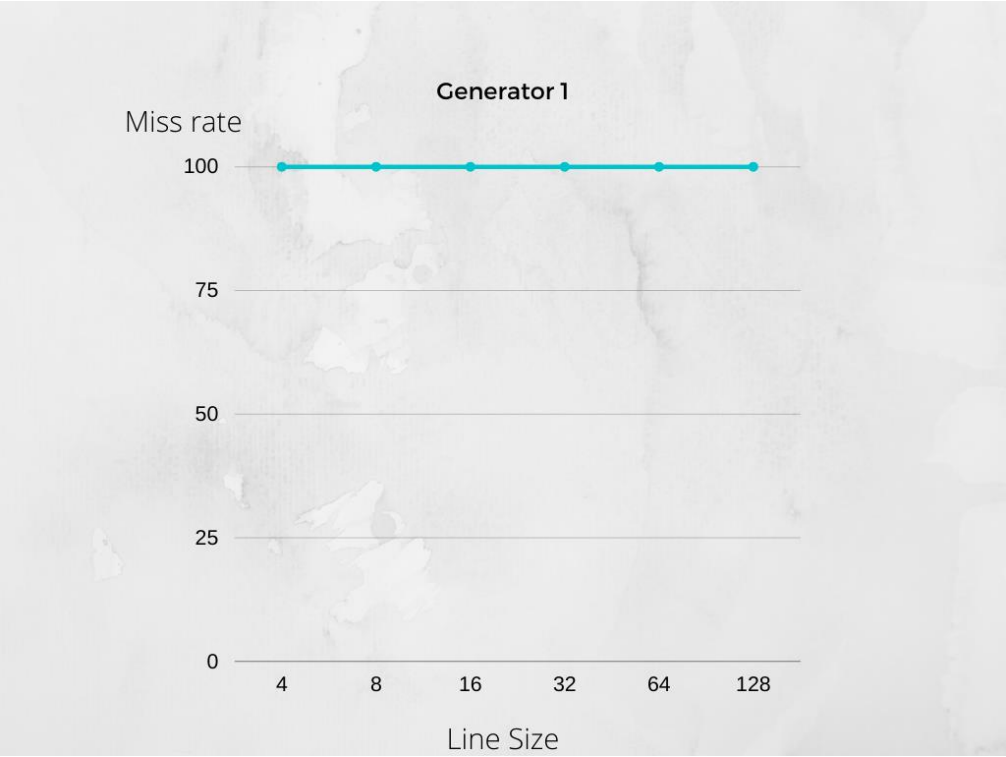
Experiment 1: How the change in Line Size affects the cache

Dependent variable: Miss rate Independent Variable: Line size. Control Variable: Cache size.

In this experiment, the effect of line size change on miss rate was measured. For each line size, the 6 address generators were tested and their results were recorded and graphed below.

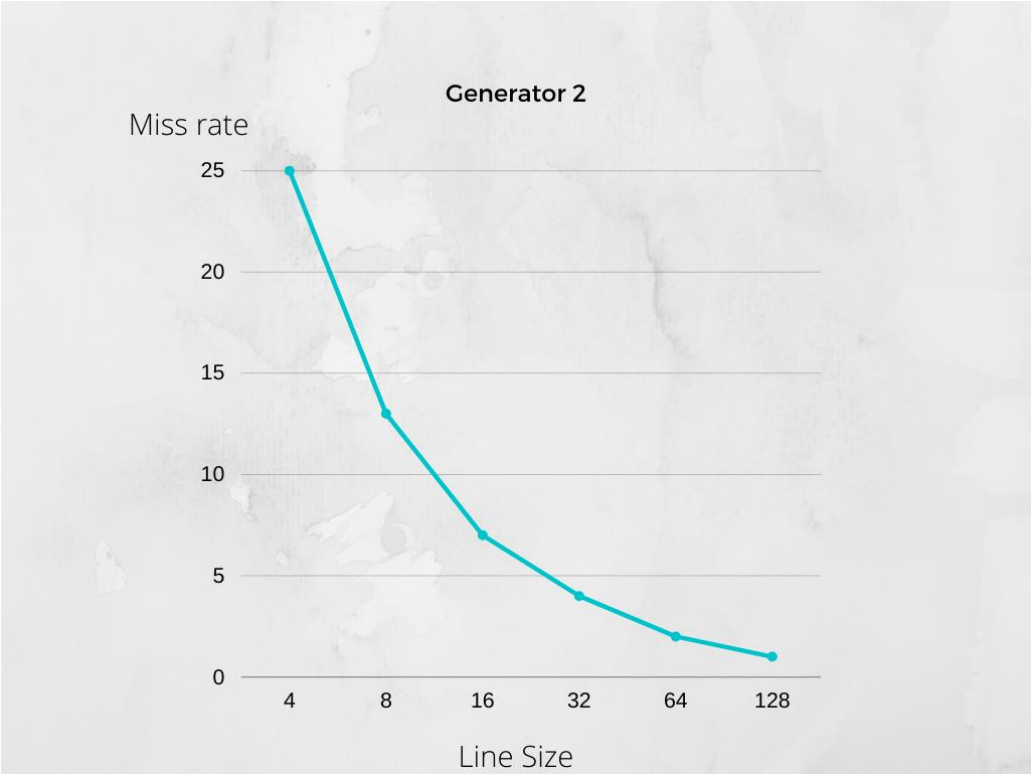
Line Size	Memgen1 Miss rate	Memgen2 Miss rate	Memgen3 Miss rate	Memgen4 Miss rate	Memgen5 Miss rate %	Memgen6 Miss rate
4	100%	25%	2%	1%	2%	100%
8	100%	13%	1%	1%	1%	100%
16	100%	7%	1%	1%	1%	100%
32	100%	4%	1%	1%	1%	100%
64	100%	2%	1%	1%	1%	100%
128	100%	1%	1%	1%	1%	51%

The graphs for the following data are presented below.



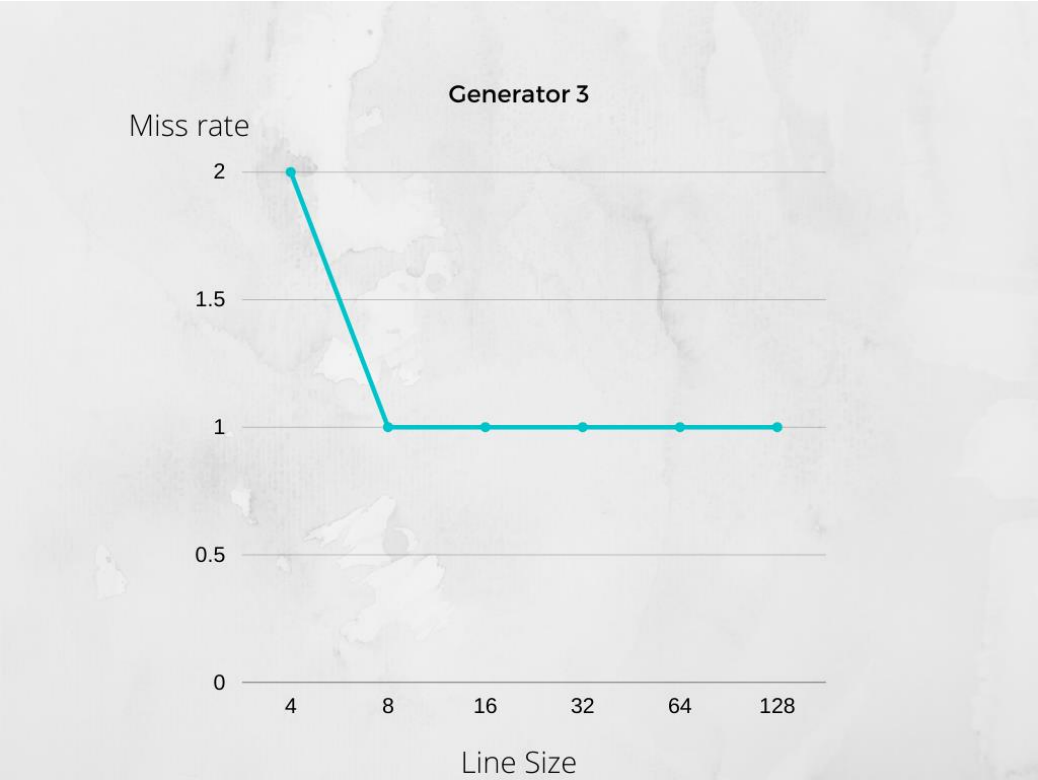
Observation: The Miss rate is constant no matter how the line size changes.

Analysis: The generator gives addresses that have a huge difference between them and all numbers will have different indices or different tags. Therefore, the miss rate will always be 100%.



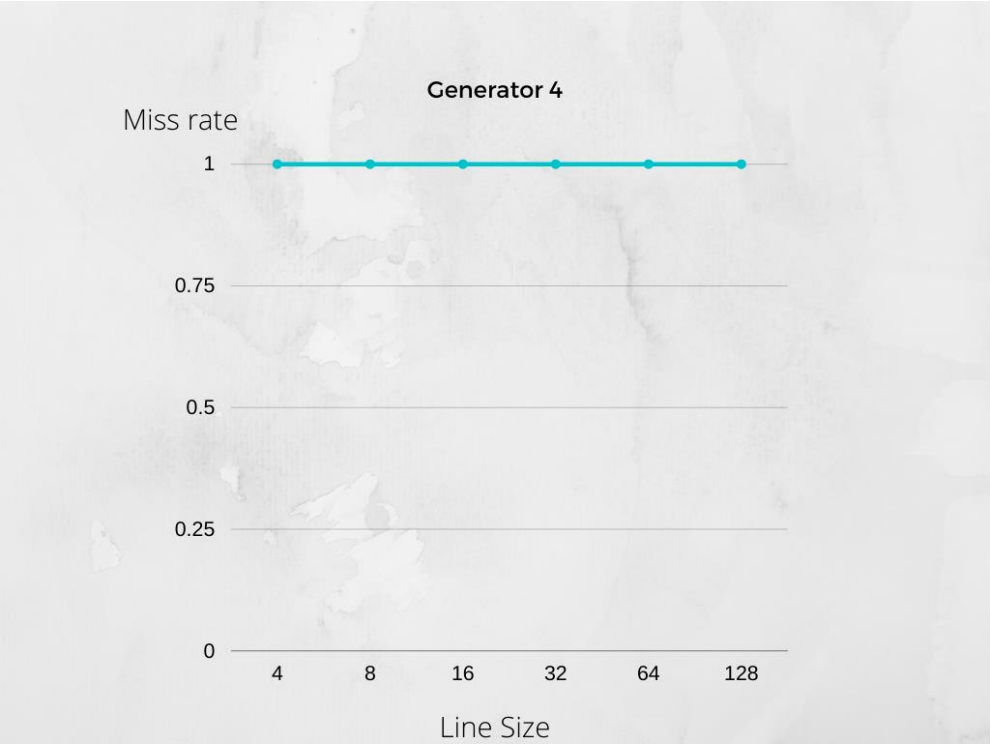
Observation: The Miss rate declines as the line sizes increases. There's an inverse relation between miss rate and line size.

Analysis: The generator gives increases the address by 1 every time it's called. Therefore, each address will be stored in a line at first and then if the same index and tag are checked again it will be a hit. The larger the line size, the more numbers that will have hits in the same index.



Observation: The Miss rate is only different when the line size is 4 bytes.

Analysis: The generator gives addresses that are random but at the range of 0 to 64*1024. In the 100000 instructions we are testing, this means that the chance of numbers having the same index and tag are high. That's why the miss rate decrease between 4 and 8. The miss ration stays at 1.



Observation: The Miss rate is constant at 1%.

Analysis: The generator generates increasing numbers from 0 to 4096. The number of blocks available is bigger than the range of the numbers. Therefore, most of the numbers will have similar indices and will score hits unless it's an empty line.

Observation: The Miss rate changes from 2 to 1 when the line size becomes 8 bytes and it stays there.

Analysis: The generator is similar to generator 3 except that it uses sequential numbers instead of random numbers. This means that the number range from 0 to 64*1024 in order. The line size of 4 has the highest number of misses because it has more indices than the other sizes. The other sizes have the same rate of 1% because it cannot get any lower.

Observation: The Miss rate is constant at 100%. Until the size becomes 128.

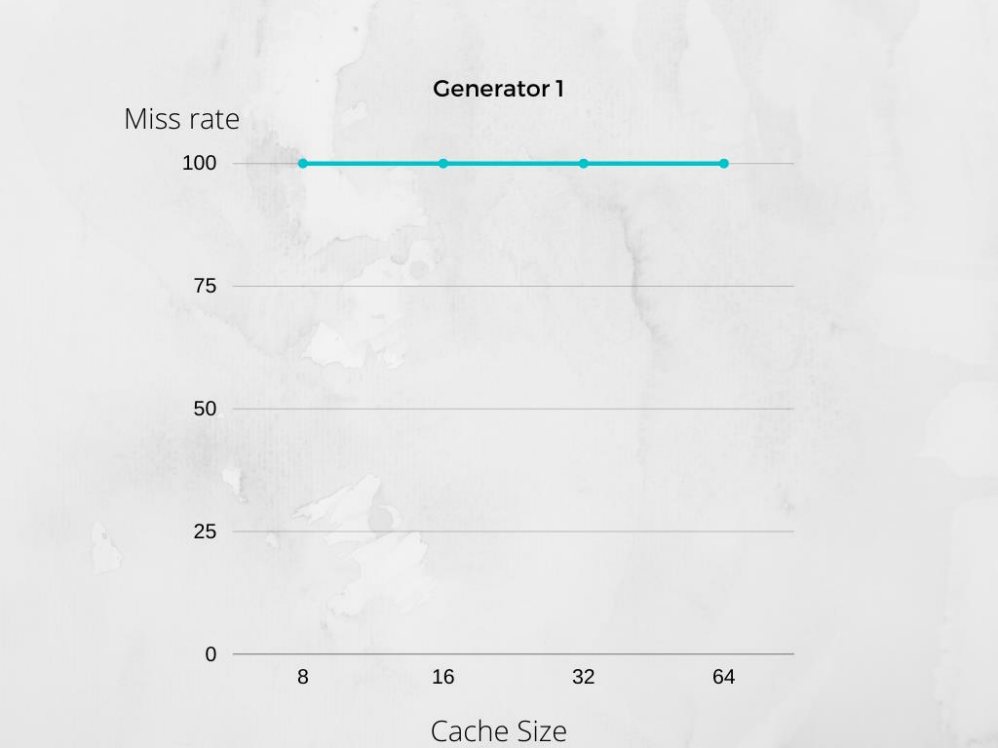
Analysis: The range of this function is between 0 and 128*1024. Therefore, the cache will not have enough blocks to contain that range and will give 100% miss rate until the line size becomes 128. Then, the cache can start giving back hits because it has space to cover all the new entries.

Experiment 2: How the change in Cache Size affects the cache

Dependent variable: Miss rate Independent Variable: Cache size. Controlled Variable: Line size.

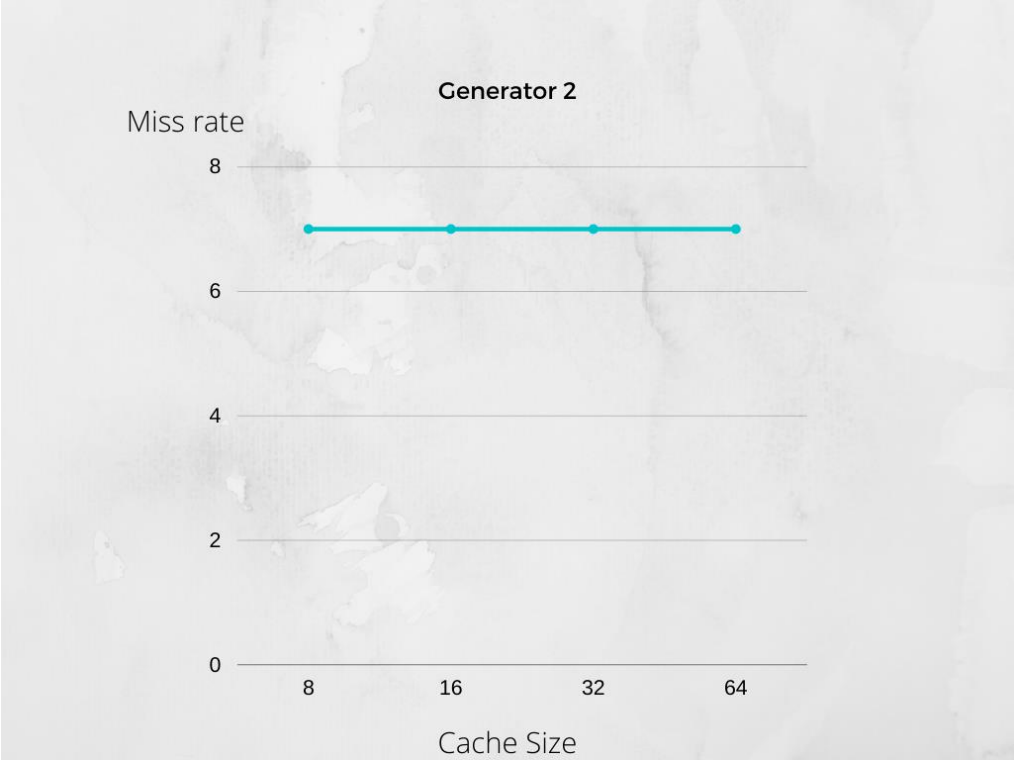
In this experiment, the effect of cache size change on miss rate was measured. For each cache size, the 6 address generators were tested and their results were recorded and graphed below. The line size is kept constant throughout the experiment with a value of 16.

Cache size	Memgen1 Miss rate	Memgen2 Miss rate %	Memgen3 Miss rate %	Memgen4 Miss rate %	Memgen5 Miss rate %	Memgen6 Miss rate %
8	100 %	7%	88%	1%	7%	100%
16	100 %	7%	75%	1%	7%	100%
32	100 %	7 %	51%	1%	7%	100%
64	100 %	7 %	1%	1%	1%	100%



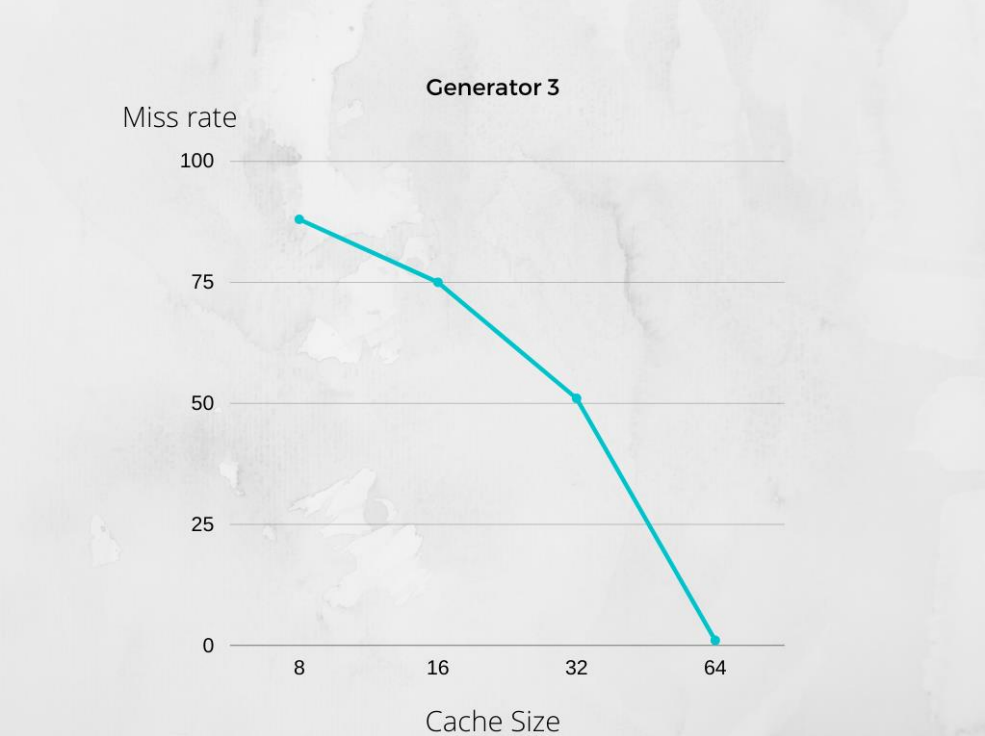
Observation: The Miss rate is constant at 100%.

Analysis: The generator generates increasing numbers from 0 to 4096. The number of blocks available is bigger than the range of the numbers. Therefore, most of the numbers will have similar indices and will score hits unless it's an empty line.



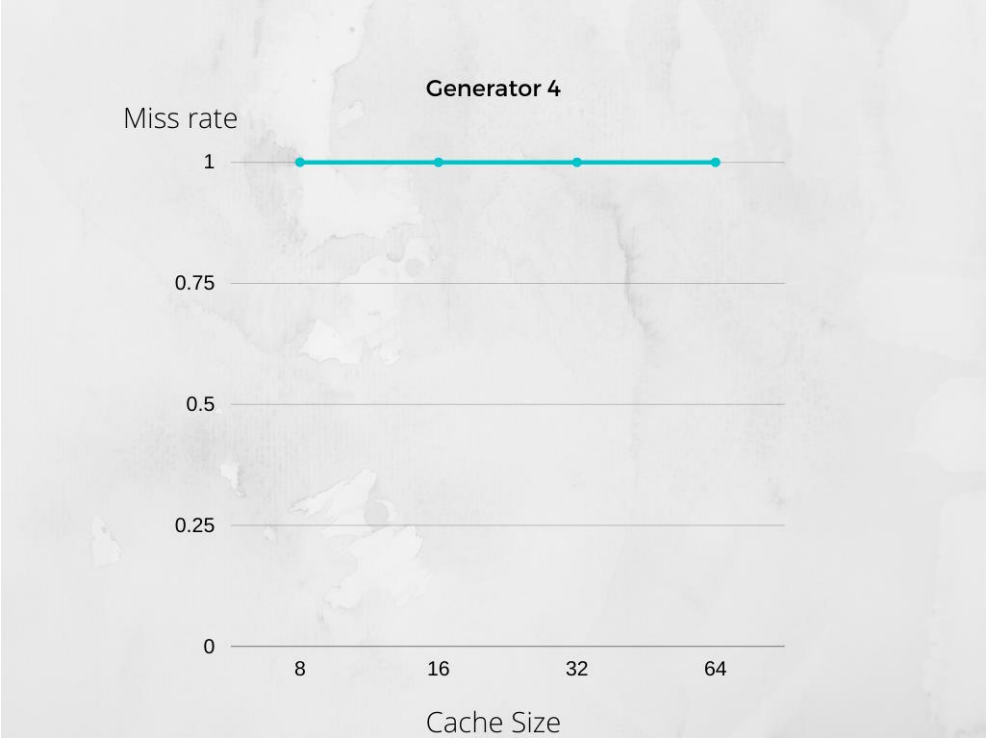
Observation: The Miss rate is constant at 7%.

Analysis: The generator generates sequential numbers from the range 0 to the D ram size. Therefore, it will first generates misses to fill the indices and after that the indices will start repeating themselves.



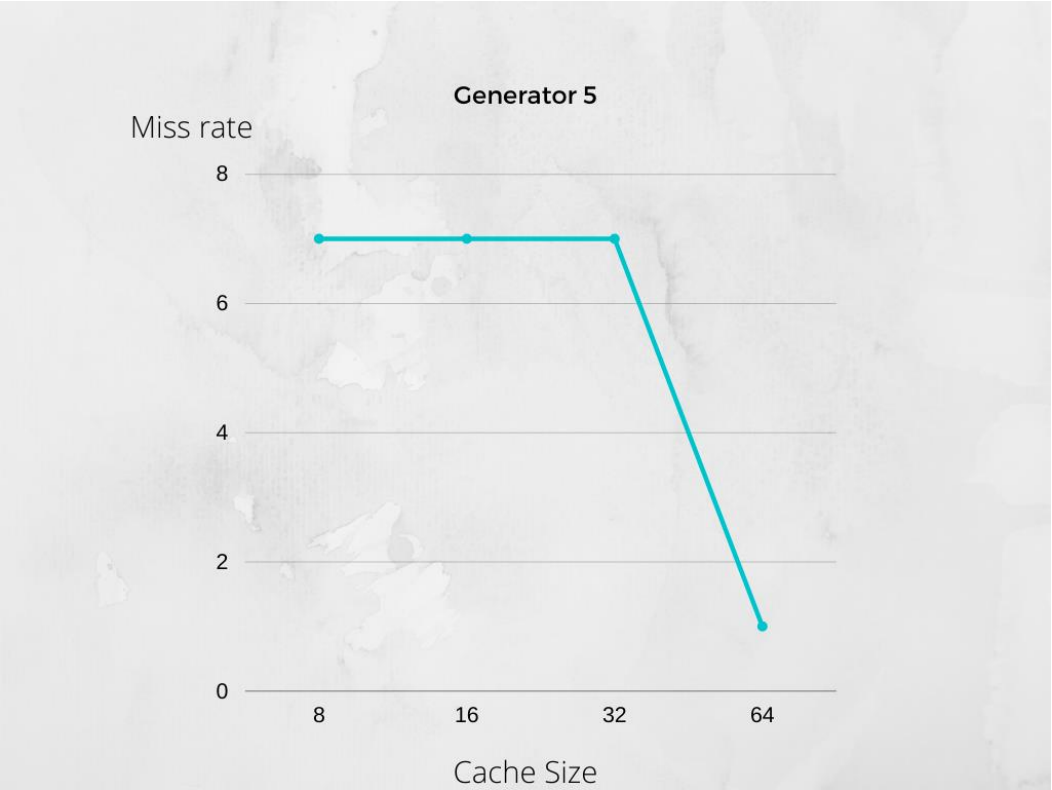
Observation: The Miss rate declines as the cache size increases.

Analysis: In this generator the numbers are generated randomly from 0 to 64*1024. And, as the cache size increases, the number of blocks increases and we can store more values in the cache, which means more hits as the size increases.



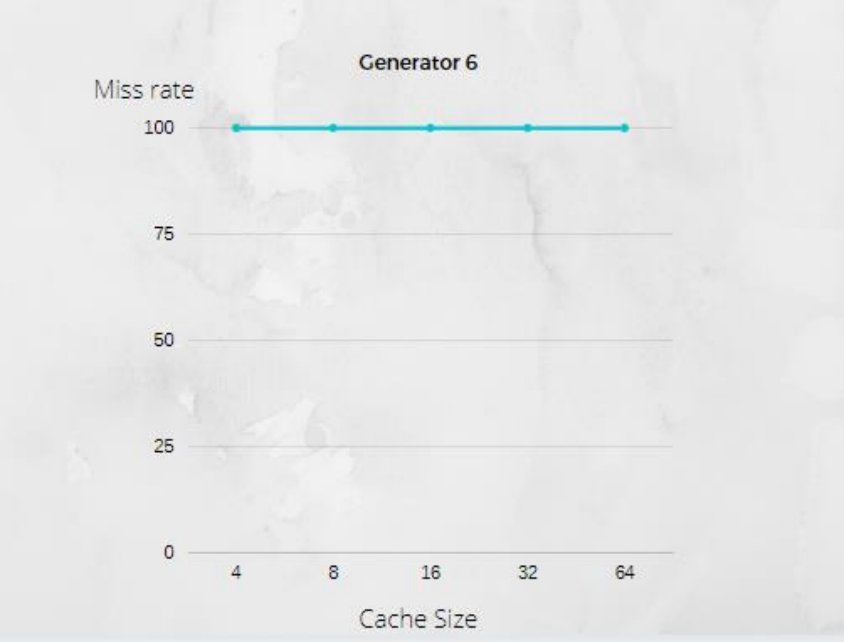
Observation: The Miss rate is constant at 1%.

Analysis: The generator generates increasing numbers from 0 to 4096. The number of blocks available is bigger than the range of the numbers. Therefore, most of the numbers will have similar indices and will score hits unless it's an empty line.



Observation: The Miss ratio is constant at 7% and changes to 1% when the cache size is 64.

Analysis: The generator is similar to generator 3 except that it uses sequential numbers instead of random numbers. This means that the number range from 0 to 64*1024 in order. When the cache size becomes 64, all the compulsory misses will be filled at first and then, any number that comes after it will be a hot. That's why the number becomes 1% at 64kb size.



Observation: The Miss rate is constant at 100%.

Analysis: The range of this function bigger than the available cache size. Therefore, every new address will result a conflict miss because it must overwrite another address stored there.

Conclusions:

Number of blocks= cache size in bytes / line size in bytes.

From the following formula we deduce that the number of blocks is directly proportional with cache size and inversely proportional with line size. However, in the two experiments, we are using generators that produce addresses that have a certain sequence. Half of the generators are sequential and the other half randomly generate numbers. Therefore, each case is different and the miss ratio is totally dependent on which generators are used.

Section 4: Generators analysis

Memgen1:

This is that this function covers a huge range of numbers. Numbers from 0 to the DRAM size which is 64*1024*1024= 67108864. This number is bigger than 1000000 and therefore, the possibility of having a hit is extremely small and that's why the miss rate was 100%.

```
unsigned int memGen1()
{
    return rand_() % (DRAM_SIZE);
}
```

Memgen2:

Theis function uses an unsigned int and increments it after each call. The use of a static int and incrementing it after each call means that the addresses will increase sequentially and therefore resulting in a low miss ratio.

```
unsigned int memGen2()
{
    static unsigned int addr = 0;
    return (addr++) % (DRAM_SIZE) ;
}
```

Memgen3:

The miss rate was variable in this function because this function uses the rand function with a range from 0 to 64*1024=65536 which is less than 100000. This means that the function can generate the same number in the range of 1000000 iteration. That is why the miss rate is variable in this function.

```
unsigned int memGen3()
{
    static unsigned int addr = 0;
    return  rand_() % (64 * 1024) ;
}
```

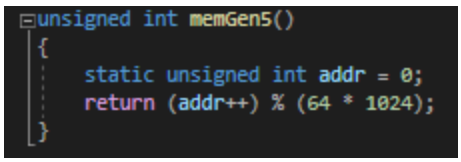
Memgen4:

This function has a constant miss rate at 1% in Experiment 2. This is because it uses a static int it and increments it after every call. The range of the function is from 0 to 1024*4=4096. The function depends on the value of the incremented variable addr.

```
unsigned int memGen4()
{
    static unsigned int addr = 0;
    return (addr++) % (1024 * 4) ;
}
```

Memgen5:

This function has a variable miss rate only at the size of 64. This is because the function is similar to memgen4 but it uses 64*1024 as the number after the modulus. The only cache size that will have numbers equivalent to that range is the 64kb cache size.



Memgen6:

The miss rate here only becomes less than 100% when the line size becomes 128 bytes. That's because the function generates addresses in the range of 0 to 128*1024. Smaller line sizes will result in endless conflict misses.